1. Introduction
2. Background
   1. Paxos

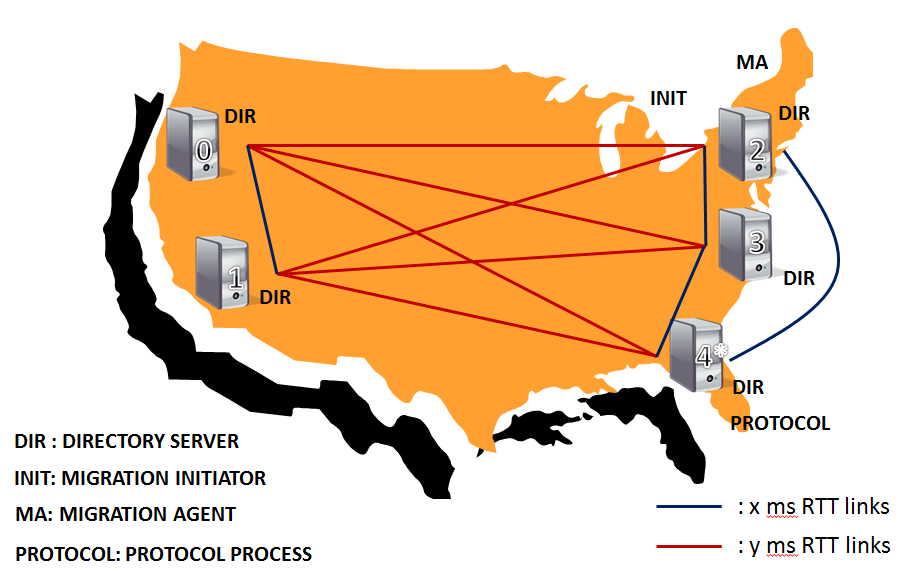
* An algorithm for implementing fault-tolerant distributed systems.
* The heart is a consensus algorithm – how do we get multiple processes that are each trying to assert/propose a value to agree upon and stick with a single value?
* The safety requirements:
  + Only a single value that has been proposed may be chosen
  + Processes learn about values if and only if they have been chosen
* The protocol does not specify any liveness/convergence requirements.
* There are 3 classes of “agents” that take part in the protocol:
  + Proposers – They propose values to be chosen
  + Acceptors – They choose to or not to accept proposed values
  + Learners – They learn the final, single proposed value that was accepted by the acceptors (not all, just a majority, see below)
  + There are no strict requirements on the mappings between the given processes and these roles
* A proposed value can be considered accepted once a majority of acceptors have accepted it.
* The cornerstone of the algorithm lies in determining how and which value must be accepted.
* From a bird’s eye perspective, the acceptors control the proposers and their proposed values – so the working of the algorithm is driven by acceptors forcing the proposers to propose acceptable values, whilst the design of the algorithm revolves around setting down rules for how to accept values.
* The design considerations for accepting values are as follows (revised as new requirements emerge:
  + 1. An acceptor must accept the first proposal it receives – we must begin somewhere
  + Only a single value must be accepted => we’ll turn this around and instead put the responsibility on the proposers and say – only that value may be proposed repeatedly
  + As a proposer, I can see what values have been accepted while proposing, but I cannot predict what values might be accepted in the future. To this end, I somehow seek to control the future acceptances by extracting promises from acceptors regarding the nature of the same
    - Proposals now have a proposal number. To avoid confusions, different proposals must have different numbers, a global ordering of sort – the implementation left open ended. A suggestion would be to just have proposers choose the numbers from non-overlapping sequences and store the last used number in stable storage.
    - Promise to me, the proposer that you, the acceptor will not accept a proposal with a number lower than mine
    - If you have already accepted a proposal, let me know.
  + Due to this extracted promise, we need to change acceptance rule 1 to: 1a. Acceptors can and must only accept proposals that do not violate promises it has made => accept proposals which have numbers > numbers of proposals to which promises have been made
  + 2. If a proposal with value ‘v’ is chosen, then every higher numbered proposal that is chosen by any acceptor has value ‘v’ – this follows from the requirement that only a single value be chosen in a round of Paxos.
  + This is where the implementation of the algorithm is driven backwards – to ensure that no proposal with a value other than ‘v’ with a proposal number higher than the highest accepted proposal number (with value ‘v’) is accepted, the acceptors force the proposer to only issue proposals with value ‘v’. Hence:   
    2a. If a proposal with value ‘v’ is chosen, then every higher-numbered proposal issued by any proposer has value ‘v’.
  + Now we relax constraint 2a by moving to a majority instead of every acceptor. Hence:   
    2b. For any proposal numbered ‘n’ with value ‘v’ issued, there exists a set ‘S’ consisting of a majority of acceptors such that either:
    - a) no acceptor in S has accepted any proposal numbered less than ‘n’, or
    - b) ‘v’ is the value of the highest numbered proposal among all proposals numbered less than ‘n’ accepted by acceptors in ‘S’
* Putting all this together, the algorithm for a single ‘round’ of Paxos sums up to such:   
  Phase 1.   
  (a) A proposer selects a globally exclusive proposal number ‘n’ and sends a prepare request to a majority of acceptors (it could be all acceptors in the implementation) – this is called a ‘prepare’ request.  
  (b) If an acceptor receives a ‘prepare’ request with number ‘n’ greater than any ‘prepare’ request to which it has already responded, it responds to the request with a promise not to accept any more proposals with number less than ‘n’, and the number ‘n’ and value ‘v’ of the highest number proposal it has accepted (if any).  
  Phase 2.  
  (a) If the proposer receives a response to its prepare request numbered ‘n’ from a majority of acceptors, then it sends an ‘accept’ request to each of those acceptors for a proposal numbered ‘n’ with either the value of the highest numbered proposal it received from the acceptors in response to its prepare request, or if no such value exists, then any value of its choosing.  
  (b) If an acceptor receives an accept request for a proposal numbered ‘n’ >= highest prepare request number it has responded to, then it accepts the proposal.
* A few things to note:
  + There is no direct correlation between Phases 1 and 2 in terms of a Phase 1 being sufficient for Phase 2. That is, a proposer ‘P1’ could elicit a response to its prepare request but it might end up racing with another proposer ‘P2’ in that acceptors could end up rejecting P1’s accept requests after accepting its prepare requests because P2 is racing P1 and keeps issuing prepare requests with numbers succeeding P1’s prepare requests.
  + A decision is implicitly reached when a majority of the acceptors accept the same value ‘v’ – because using induction and the property of there being at least one common acceptor in the intersection of 2 majorities of acceptors, we can show that the acceptors will force any future proposers into re-proposing the same accepted value.
  + There is no limit on the number of proposals that can be made – proposers can abandon proposals mid-flight and reissue proposals of higher numbers if they want.
  + There is no guarantee of convergence – the protocol is correct, but may never converge.
* To learn a chosen value, the learners must find out that a majority set of acceptors have accepted a single value. There are multiple ways to do this, the most straightforward of which would be to have every acceptor ack acceptances it makes to every learner.
* Optimizations:
  + The first obvious optimization would be some step to alleviate the non-convergence problem. We could have a “distinguished proposer”, a leader who would be the only one trying to issue proposals, circumventing the race problem.
  + Similarly, we could have a distinguished learner, or a set of them to reduce the number of acks that the acceptors would have to send out once they accept a value.

1. Design and Implementation
   1. Dexter and DTunes?
   2. The Migration Service
      * The part of the DTunes project that we decided to implement was the directory service.
      * This service needed to be a fault tolerant service which keeps track of object migrations initiated by the compute client.
      * What we are guaranteeing here overall is that once a migration request has been saved by the Paxos members, they will eventually complete the migration.
      * We separate the migration process into 4 main concerns:
        + A directory service that implements the Paxos service interface fronted by the JPaxos code
        + A migration protocol that takes care of the actual migration process
        + Directories that store the current location of a given object
        + Migration Agents that perform the actual process of copying the object from the source to the destination location.
   3. The Directory Service
      * The Directory Service implements the service wrapper that JPaxos fronts.
      * This means, apart from the actual execution of decided Paxos requests themselves, it also handles concerns like snapshotting and restoration from snapshots.
      * What we had to be cognizant of in the design of the service is that there is no single replica that can be in charge of the migration of even just a single object.
      * Reason for this being we can have failures and leader changes mid-migration.
      * When a new object migration is received (and decided upon), a record is created in the database to represent the migration request.
      * This record represents all the state information associated with the object.
      * The fields stored in the record are:
        + ObjectId: The ID of the object being migrated
        + Old Replica Set: Where the object is currently being stored
        + New Replica Set: Where the object will be stored after the migration
        + Directory Acks: Directories that know about the new location of the object
        + Migration Progress Acks: Migration agents that have ACK’d back, having completed their share of the object movement
        + Creation Time: Time of creation of the record
        + Completion Time: Time the migration was completed
        + Last Updated Time: Time this last record was last serviced (read/updated). This is meant to be a mechanic to avoid starvation.
        + Migration Started Timestamp: Time the Migration Agents were informed about the required object movement.
        + Migrated: Boolean state of whether the object has been moved from its old replica set to the new one.
        + Migration Complete: Boolean state of whether the whole migration process has been completed for this object.
      * These fields are updated as the object moves through the migration process, and are also used to make decisions about what must be done next to complete the migration for the object.
      * Since we are implementing a fault-tolerant service, we can see that all write operations (insert and update) made on the record must be Paxos operations.
      * The first thing we had to consider while implementing the service was that the way the Paxos protocol as implemented recovers from a failure is by replaying batches of requests.
      * Depending on the crash recovery model we use, the strategy of recovery changes.
      * The EpochSS recovery model that we used replays all requests from the last saved Epoch, not just saved requests.
      * This means that decided requests that the crashed replica played before crashing will now be replayed.
      * JPaxos has been designed to replicate services that have no state surviving  
        crash.
      * However, what JPaxos does give us is a sequence number for every request that has the following properties:
        + It is monotonically increasing
        + It is consistent amongst replicas (the same request will have the same sequence number for every replica)
        + There are no gaps between 2 numbers
      * Using this primitive, we can design a stateful replication service.
      * We save the sequence number of the last executed request in the database and when we replay requests, we skip all requests up till the last executed one.
      * One important thing to note here is that the saving of the last executed request’s sequence number and the effect of the last executed request itself (in terms of any database operations) must be one single atomic operation.
      * Now that we have a stateful replicated system, the next thing to discuss would be the operations that the service supports:
        + Insert: creates a migration record in the database. By design, only one outstanding migration is allowed for an object. This design choice has tight correlation with the design choice of optimizing local database reads as opposed to Paxos reads in the migration protocol.
        + Update operations: updates on the above listed fields of the migration record. In the directory service, updates are triggered by either the protocol process or a migration agent.
        + Read: reads and serves the current migration state for the given object id
        + Delete: deletes the migration record for the given object from the database
        + Register operations: register directories/migration agents that are bootstrapping
      * To implement snapshotting, instead of restoring from a snapshot by replaying all requests contained in the snapshotted period, we snapshot the state of the database itself.
      * This makes the implementation straightforward and in some cases we end up with fewer database operations this way.
      * An example of such a case would be a snapshot of a single completed migration’s database record – if we maintained a traditional transaction based log, we would have about 8 database write transactions to reach completion. If on the other hand we just snapshotted the finished database state, we would achieve the same effect, in a single insert statement.
      * When restoring from a snapshot, the restoring replica wipes its database state clean and completely restores the state received.
   4. The Migration Protocol
      * The protocol is a process that runs co-hosted with all the replicas
      * The only protocol process that can make any decisions and take actions is the one co-hosted with the leader process.
      * Reason for this being, the protocol process co-hosted with the leader makes DB local reads (as an optimization instead of making Paxos reads)
      * When a leader change occurs, the process co-hosted with the new leader seamlessly picks up from where the old one left off.
      * To do this, we have a polling mechanism set up between the replica and protocol processes.
      * Since we want the protocol processes to be able to function in this seamless fashion without any communication between them on failure, the main design consideration was statelessness.
      * The protocol process is essentially a state machine. It picks up a few objects which still have been slated for migration and are in different stages of their migration and pushes them to completion.
      * The process is not currently threaded – that is it only performs one migration at a time.



A step-by-step progress of the migration process

* + - Each migration entry when created
  1. Logging Framework

1. Experimental Methodology and Results
   1. Aim
      * The high level aim of the experiments was to get a deeper understanding of the Paxos algorithm in an actual implementation.
      * This effort can be broadly classified into 2 steps:
        + Understanding the implementation and it’s departures from the algorithm in terms of optimizations and details generally left open to implementation.
        + Instrumenting the implementation to piece-wise analyze the different component times in the algorithm.
      * Coupling this with different delays simulated between replicas would give us an example of an instrumented Paxos system deployed in a WAN.
      * The Directory Protocol gives us a working state machine system built around the Paxos core to drive the experiments.
   2. Setup  
      The experimental is as shown below:  
        
      

* We used a 5 replica cluster.
* The cluster was split into 2 sub-clusters. Depending on the inter and intra cluster delays this allowed us to model different geographical setups.
* Each replica runs its own directory server
* The Migration Initiator is co-located with the Migration Agent on one of the replicas. We did not want to end up needing a lot of the machines to conduct these experiments. Also since all 3 processes are rather lightweight, there would be limited overhead/contention between them.
* The Protocol Process runs on every machine, but since for the purpose of these experiments we are not simulating any failures, all processes apart from the one co-located with the leader replica are of no consequence as they would be unable to take any action.
* Since the focus here is on latency and not throughput measurements, we set the parameters to have high polling frequencies, and low critical path latencies.
  + CrashModel – EpochSS
  + WindowSize – 2 (the migrations are serial for the purpose of these experiments)
  + MaxBatchDelay – 0 (do not batch, push proposals instantly on arrival)
* The implementation had been modified to force any Paxos client to always connect to the leader process (directly, no redirections) and the same replica (#4) is elected leader for all experiments. This forces uniformity between runs so we can set strong expectations for our experiments.
  1. Experiments
     + The experiments are each run for 10 key migrations – each key migration involves 5 Paxos stages, namely:
       - Migration Initiation
       - Directory Acks – 1 per agent (5 in our implementation)
       - Updating Timestamp after informing Migration Agent about object move
       - Migration Agent Acks – 1 per agent (1 in our implementation)
       - Updating migration record to complete
     + All graphs are plotted on the basis of above mentioned Paxos rounds.
     + We ran our experiments for 3 different network configurations. Discussed below are each configuration and associated expectations and results.
     + No DummyNet:
       - These set of experiments are running on the PRObE testbed’s Infiniband fabric directly.
       - The link delays between nodes i.e. x = y = ~0.1ms.
       - Expectations:
         1. The Paxos latencies as measured from the enqueued phase to decide phase measured at the leader (as that is the only replica which has to power to queue and propose) should be of the order of a few milliseconds (~1 RTT). This is heavily dominated mainly by 2 factors:

Code execution time and threading overhead.

Queuing latencies between the multiple asynchronous parts of the application

* + - * 1. The client end to end latencies should be dominated by the service time of the request. The link latencies are almost negligible and the Paxos part of the request servicing should only come out to a few milliseconds. The service time again can be splits into 2 dominant factors:

Code execution time of the state machine itself

Database access times

* + - * Results:



Paxos client end to end latency



Paxos leader latency



Directory service time

* While we have some deviation from the expected case, if we round-wise sum the service times and Paxos leader latencies, they line up with the Client end to end latencies.
* The client end to end latencies may not be uniform because of the different natures of the Paxos clients:
  1. Migration Initiator and Migration Agent processes run on different machines from the leader process, and will hence experience link delays.
  2. The Paxos client for the other rounds is the Protocol process which is running co-hosted with the leader replica, and will only experience inter-process communication delay.
* The possible mismatches could be due to a few reasons such as:
  1. The client end to end latency includes a round trip from the paxos client to the leader replica and whilst on inifniband this link latency itself is a small value, there is a finite queuing time involved in the transmission of the message within the application itself.
  2. + DummyNet with 0ms delay
       - DummyNet introduces a finite amount of overhead. If we configure DummyNet to emulate link delays to be 0ms, there is still a finite ping time between nodes.
       - This time was roughly observed to be around 5ms.
       - Plotting this case will now give us a benchmark for an overhead over our pure inifniband case.
       - The expectations here are the same as the no dummynet case and any observed deviation will be treated as the dummynet overhead.
       - Results:



Client end to end latency



Paxos leader latency



Directory service time

* Now we notice that we have a large deviation from the expected case (which would be that these graphs be very similar to the no dummynet graphs)
* Since there has been no change between these 2 experiments apart from dummynet being introduced to model 0ms delay, all of the deviation can be attributed to dummynet overhead.
* Now as the Paxos leader latency times represent ~ 1 RTT of link delay, we can use the overhead in that graph to fit and verify our earlier proposed explanation to the mismatch between the round-wise sum of the Directory service times + Paxos leader latencies and the Client end to end latency
  1. The client end to end latency includes a round trip from the paxos client to the leader replica.
  2. The earlier explanation attributed the extra time to message queuing delays in the application, but with this new overhead we are seeing due to dummynet for the Paxos leader times (~1 RTT) would now make the round trip from the client to the leader replica significant and in fact the dominant factor.
  3. Accounting for this, our observations line up as expected – Paxos leader latencies + Directory service times + Paxos leader latencies representing client-leader roundtrip ~= Client end to end latencies.  
       
     + Dummynet with uniform 20ms (RTT) latency
       - Now we simulate the 5 nodes being at equal link delays of 20ms from each other.
       - i.e. x = y =20ms
       - Expectations:
         1. The convergence time of a single Paxos round measured at the leader would now be composed of 3 factors:

Code execution time and threading overhead.

Queuing latencies between the multiple asynchronous parts of the application

Since all replicas are 20ms (RTT) away from the leader, ~1 RTT convergence time for the Paxos round would also now be expected to kick in.

* + - * 1. Since in our baseline pure Infiniband only experiment we saw Paxos leader times of ~3-5ms which account for i. and ii., overall we would expect to see a 20 + 3-5ms delay.
        2. But we also note that in our 0ms dummynet experiment that we saw a ~20ms overhead in our Paxos leader latencies.
        3. So while we would ideally expect to see a 25ms Paxos leader latency graph, it would be offset by the overhead, moving up our expectations to around 45ms.
        4. The client end to end latencies can be split into dominant factors:

The client-leader RTTs for the initiation and migration agent ack rounds – ~20ms

The service time for the request – DB + code execution time

The Paxos time itself - ~20ms (1 RTT)

* + - * 1. Hence we expect a 20ms (for initiation and migration agent acks) + 20ms + 3-5ms (baseline Paxos) + 20ms dummynet overhead + DB + code execution time.
      * Results:



Client end to end latency



Paxos leader latency



Directory service time

* We notice that most of our expectations were right.  
  + - Dummynet with 20 and 80ms delays
      * We simulate a setup with 3 machines on the east coast and 2 on the west coast.
      * x = 20ms, y = 80ms.
      * Expectations:
        1. The convergence time of a single Paxos round measured at the leader would now be composed of 3 factors:

Code execution time and threading overhead.

Queuing latencies between the multiple asynchronous parts of the application

The convergence time for a single Paxos round would still be expected to be around 20ms as the leader replica has the ability to form a majority with the 2 other east coast replicas.

* + - * 1. Other expectations remaining the same as the previous case, we would still ideally expect to see a 25ms Paxos leader latency graph, it would be offset by the overhead, moving up our expectations to around 45ms.
        2. The client end to end latencies can be split into dominant factors:

The client-leader RTTs for the initiation and migration agent ack rounds – ~20ms

The service time for the request – DB + code execution time

The Paxos time itself - ~20ms (1 RTT)

* + - * Hence we expect a 20ms (for initiation and migration agent acks) + 20ms + 3-5ms (baseline Paxos) + 20ms dummynet overhead + DB + code execution time.
      * Results:



Client end to end latency



Paxos leader latency



Directory service time

* Our Paxos leader times as expected are still around the 45ms mark – validates our claim about the east coast majority causing consensus.
  + - * Other results match up as expected.
    - Something that we observe across the service time results above is that the initiation and migration agent acks times seem to be consistently higher than the other 3. This can be attributed to the following reasons:
      * Database – Initiation is an insert operation. The other operations are all update operations. This leads us to believe that the DB optimizer somehow seems to favor performance of one over the other.
      * Migration Agent Acks by design is a 3 step database operation due to its asynchronous nature.
        1. First the migration agent acking is identified through its IP and port information.
        2. Then the current migration progress for the object being migrated is looked up.
        3. Finally, if this is an ack from an agent that hasn’t already acked, the entry is recorded.

1. Conclusion